Races and Deadlocks

COMS W4118
Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s
Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
• Identify **patterns of concurrency errors**
  – so you can avoid them in your code

• Learn **techniques to detect concurrency errors**
  – so you can apply these techniques to your code


Concurrency error classification

- **Deadlock**: a situation wherein two or more processes are never able to proceed because each is waiting for the others to do something
  - Key: circular wait

- **Race condition**: a timing dependent error involving shared state
  - **Data race**: concurrent accesses to a shared variable and at least one access is a write
  - **Atomicity bugs**: code does not enforce the atomicity programmers intended for a group of memory accesses
  - **Order bugs**: code does not enforce the order programmers intended for a group of memory accesses
Examples

- **Deadlock**
  
  T1
  
  lock(m1);
  
  lock(m2);
  
  lock(m2);
  
  lock(m1);

  T2

- **Data race**
  
  ++ balance

  --balance

- **Atomicity**
  
  len = 100;
  
  buf = realloc(len);

  if (len > 200)
  
  memcpy(buf, str, 200);

- **Order**
  
  p = NULL

  *p;
Benign race examples

• Double-checking locking
  – Faster if \( v \) is often 0
  – Doesn’t work with compiler/hardware reordering

```c
if(v) {
  // race
  lock(m);
  if(v)
    ...
  unlock(m);
}
```

• Statistical counter
  – ++ nrequests
Writing correct parallel code is hard!

- **Too many** schedules (exponential to program size), hard to reason about

- Correct parallel code does not compose → can’t divide-and-conquer
  - Synchronization cross-cuts abstraction boundaries
  - Local correctness may not yield global correctness.

- We’ll see a few error examples next
Example 1: good + bad ⇒ bad

- Result: race between deposit() and withdraw()

  deposit() // properly synchronized
  lock();
  ++ balance;
  unlock();

  withdraw() // no synchronization
  -- *balance;
Example 2: good + good → bad

- Compose single-account operations to operations on two accounts
  - deposit(), withdraw() and balance() are properly synchronized
  - sum() and transfer()? Race

```c
void deposit(Account *acnt)
{
    lock(acnt->guard);
    ++ acnt->balance;
    unlock(acnt->guard);
}

int balance(Account *acnt)
{
    int b;
    lock(acnt->guard);
    b = acnt->balance;
    unlock(acnt->guard);
    return b;
}

void withdraw(Account *acnt)
{
    lock(acnt->guard);
    -- acnt->balance;
    unlock(acnt->guard);
}

void transfer(Account *a1, Account *a2)
{
    withdraw(a1);
    deposit(a2);
}

int sum(Account *a1, Account *a2)
{
    return balance(a1) + balance(a2)
}
```

Example 3: good + good ➔ deadlock

- 2\textsuperscript{nd} attempt: use locks in \texttt{sum()}
- One \texttt{sum()} call, correct
- Two concurrent \texttt{sum()} calls? Deadlock

```c
int sum(Account *a1, Account *a2)
{
    int s;
    lock(a1->guard);
    lock(a2->guard);
    s = a1->balance;
    s += a2->balance;
    unlock(a2->guard);
    unlock(a1->guard);
    return s
}
```

T1: sum(a1, a2)  
T2: sum(a2, a1)
Example 4: monitors don’t compose as well

- Usually bad to hold lock (in this case Monitor lock) across abstraction boundary

```c
Monitor M1 {
    cond_t cv;
    foo() {
        // releases monitor lock
        wait(cv);
    }
    bar() {
        signal(cv);
    }
};

Monitor M2 {
    f1() {M1.foo();}
    f2() {M1.bar();}
};

T1: M2.f1();
T2: M2.f2();
```
Outline

• Concurrency error patterns

• Concurrency error detection
  – Deadlock detection
  – Data race detection
Automatic software error detection

• Static analysis: inspect the code/binary without actually running it
  – E.g., gcc does some simple static analysis
    • $ gcc –Wall
• Dynamic analysis: actually run the software
  – E.g. valgrind
    • $ valgrind run-test

• Static v.s. dynamic
  – Static has better coverage, since compiler sees all code
  – Dynamic is more precise, since can see all values

• Which one to use for concurrency errors?

• Runtime detection
  – Detect problems when they happen in production
  – Cannot prevent only recover
A Historical Perspective on Deadlocks

• Deadlock handling is a problem once beloved of computer science theorists
  – Many deadlock avoidance/detection techniques in the literature

• Canonical Example
  – Dining Philosophers
Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
  - Shared data: Rice (data set), lock chopstick [n]
- What happens if each one does Pick(left) before Pick(right)?
Deadlocks in Practice

• Ensure that the system will *never* deadlock
  – Easy to do by ordered locking, but programmers forget
  – Harder in the kernel – some code can’t be preempted

• Allow the system to deadlock and then recover
  – Hard to do, recovery can be application specific

• In reality: ignore the problem and let applications deal with it; used by most operating systems, including UNIX
  – OS still cares about deadlocks *within the kernel*
From the kernel source tree: kernel/pid.c

/*
 * Note: disable interrupts while the pidmap_lock is held as an
 * interrupt might come in and do read_lock(&tasklist_lock).
 *
 * If we don't disable interrupts there is a nasty deadlock between
 * detach_pid()->free_pid() and another cpu that does
 * spin_lock(&pidmap_lock) followed by an interrupt routine that does
 * read_lock(&tasklist_lock);
 *
 * After we clean up the tasklist_lock and know there are no
 * irq handlers that take it we can leave the interrupts enabled.
 * For now it is easier to be safe than to prove it can't happen.
 */
Why do deadlocks occur?

Deadlocks can arise if the following 4 conditions hold at once:

• **Mutual exclusion:** only one process at a time can use a resource

• **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

• **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

• **Circular wait:** there’s a set \{A, B, C, ..., X\} of waiting processes such that A is waiting for a resource held by B, B is waiting for a resource that is held by C, and X is waiting for a resource held by A

Here, resources can be anything, but in practice, usually locks
Dealing with Deadlocks

• Deadlock prevention
  – Always acquire locks in same order
    • Dining philosophers: first acquire left and then right? No!
    • Doesn’t work when you can’t sleep, e.g., Interrupt handler
    • Easy to do in userspace, need best practices

• Deadlock detection
  – Detect a deadlock after it has happened, and recover

• Deadlock avoidance
  – Basic idea: detect unsafe states that might lead to deadlock
  – Often need additional information about what processes will need what resources in the future
Deadlock detection

• Root cause of deadlock: circular wait

• Detecting deadlock manually: system halts
  – Can run debugger and see the wait cycle

• Detecting deadlock automatically: resource allocation graph

• Detecting potential deadlocks automatically: lock order
Resource allocation graph

- **Nodes**
  - Locks (resources)
  - Threads (processes)

- **Edges**
  - **Assignment edge**: lock→thread
    - Removed on unlock()
  - **Request edge**: thread→lock
    - Converted to assignment edges on lock() return

- **Cycles**: deadlock

- **Problem**: can we detect potential deadlocks before we run into them?
Detecting potential deadlocks

• Can deduce **lock order**: the order in which locks are acquired
  – For each lock acquired, order with locks held
  – **Cycles in lock order** → potential deadlock

T1:
- \text{sum}(a1, a2) // locks held
- lock(a1->guard) // {}
- lock(a2->guard) // \{a1->guard\}

T2:
- \text{sum}(a2, a1) // locks held
- lock(a2->guard) // {}
- lock(a1->guard) // \{a2->guard\}

Cycle → Potential deadlock!
Multi-Resource Resource Allocation Graphs

- Cycle and deadlock
Multi-Resource Resource Allocation Graphs

- Cycle but no deadlock
Basic Idea

• If graph contains no cycles ⇒ no deadlock

• If graph contains a cycle ⇒
  – if only one instance per resource type, then deadlock
  – if several instances per resource type, possibility of deadlock
  – Use Banker’s algorithm and variants
Banker’s Algorithm

- Designed by Dijkstra for THE multiprogramming system, 1968

- Multiple instances of resources

- Each process must a priori claim maximum use

- When a process gets all its resources it must return them in a finite amount of time

- Check if an allocation is safe and won’t lead to a deadlock – i.e., there is some way to satisfy all future demands for resources
Safe States

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
Banker’s Algorithm Variables

n: processes, and m: resource types

- **Available[m]**: how many resources of type m available

- **Max[n, m]**: total number of m type resources process n will eventually need

- **Allocation[n,m]**: how many m type resources n already has

- **Need[n,m]**: how many more m type resources does n needs
  - (Max[n,m] – Allocation[n,m])
Safety Algorithm

Basic idea: check if available resources are sufficient to satisfy all future demands for all processes in some order. I.e., we are in a safe state.

1. Let Work[m] be the hypothetical future availability for resource type m, and CanFinish[n] be true if process n can finish. Initial:
   
   Work = Available
   Finish[n] = false for all n

2. Find an i such that both:
   (a) CanFinish[i] = false
   (b) Need_i ≤ Work // needs fewer resources than available

   If no such i exists, go to step 4

3. Work = Work + Allocation[i] // i satisfied: will eventually release its resources
   Finish[i] = true
   go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

- Check if new allocation request will lead to safe state before granting
- Let Max=Request to detect if we are already in deadlock
Outline

• Concurrency error patterns

• Concurrency error detection
  – Deadlock detection
  – Data race detection
Race detection

• We will look at only data race detection
  – Techniques exist to detect atomicity and order bugs, but we won’t discuss them in this class

• One approach to data race detection
  – Lockset algorithm
  – Other techniques exist in literature
Happens-before definition

• Event A **happens-before** event B if
  – B follows A in the same thread
  – A in T1, and B in T2, **and a synchronization event C such that**
    • A happens in T1
    • C is after A in T1 and before B in T2
    • B in T2
Happens-before race detection

• Tools before eraser are based on happens-before

• Sketch
  – Monitor all data accesses and synch operations
  – Watch for
    • Access of v in thread T1
    • Access of v in thread T2
    • No synchronization operation between the accesses
    • One of the accesses is write
Problems with happens-before

• Problem I: expensive
  – Requires per thread
    • List of accesses to shared data
    • List of synch operations

• Problem II: false negatives
  – Happens-before looks for actual data races (moment in time when multiple threads access shared data w/o synchronization)
  – Ignores programmer intention; the synchronization op between accesses may happen to be there

\[
\begin{align*}
T1: & \quad +y \\
& \quad \text{lock}(m) \\
& \quad \text{unlock}(m) \\
T2: & \quad \text{lock}(m); \\
& \quad \text{unlock}(m); \\
& \quad +y;
\end{align*}
\]
Eraser: a different approach

• Idea: check invariants
  – Violations of invariants ➔ likely data races

• Invariant: the locking discipline
  – Assume: accesses to shared variables are protected by locks
  – Every access is protected by at least one lock
  – Any access unprotected by a lock ➔ an error

• Problem: how to find out what lock protects a variable?
  – Linkage between locks and variables undeclared
Lockset algorithm: infer the locks

- **Intuition**: it must be one of the locks held at the time of access

- $C(v)$: a set of candidate locks for protecting $v$

- Initialize $C(v)$ to the set of all locks

- On access to $v$ by thread $t$, refine $C(v)$
  - $C(v) = C(v) \cap \text{locks\_held}(t)$
  - If $C(v) = \emptyset$, report error

- Sounds good! But ...
Implementing eraser

• Binary tool
  – Pros: does not require source
  – Cons: lose source semantics
    • Track memory access at word granularity

• How to monitor memory access?
  – Binary instrumentation

• How to track lockset efficiently?
  – A shadow word for each memory word
  – Each shadow word stores a lockset index
  – A table maps lockset index to a set of locks
  – Assumption: not many distinct locksets
Problems w/ simple lockset algorithm

• Initialization
  – When shared data is first created and initialized

• Read-shared data
  – Shared data is only read (once initialized)

• Read/write lock
  – We’ve seen it last class
  – Locks can be held in either write mode or read mode
• When shared data first created, only one thread can see it ➔ locking unnecessary with only one thread

• Solution: do not refine $C(v)$ until the creator thread finishes initialization and makes the shared data accessible by other threads

• How do we know when initialization is done?
  – We don’t …
  – Approximate with when a second thread accesses the shared data
Read-shared data

• Some data is only read (once initialized) ➔ locking unnecessary with read-only data

• Solution: refine $C(v)$, but don’t report warnings
  – Question: why refine $C(v)$ in case of read?
  – To catch the case when
    • $C(v)$ is {} for shared read
    • A thread writes to $v$
State transitions

- Each shared data value (memory location) is in one of the four states

Virgin

Exclusive

Shared

Shared/Modified

write, first thread

write, new thread

Read, new thread

Refine C(v), no check

Refine C(v) and check

write

write
Read-write locks

- Read-write locks allow a single writer and multiple readers

- Locks can be held in read mode and write mode
  - read_lock(m); read v; read_unlock(m)
  - write_lock(m); write v; write_unlock(m)

- Locking discipline
  - Lock can be held in some mode (read or write) for read access
  - Lock must be held in write mode for write access
    - A write access with lock held in read mode ➔ error
Handling read-write locks

• Idea: distinguish read and write access when refining lockset

• On each read of v by thread t (same as before)
  – C(v) = C(v) ^ locks_held(t)
  – If C(v) = {}, report error

• On each write of v by thread t
  – C(v) = C(v) ^ write_locks_held(t)
  – If C(v) = {}, report error
• Eraser works
  – Find bugs in mature software
  – Though many limitations
    • Major: benign races (intended races)

• However, slow
  – Monitoring each memory access: costly, 10-30X slowdown
  – Can be made faster
    • With static analysis
    • Smarter instrumentation (e.g., sampling)

• Lockset algorithm is influential, used by many tools
  – E.g. Helgrind (a race detection tool in Valgrind)